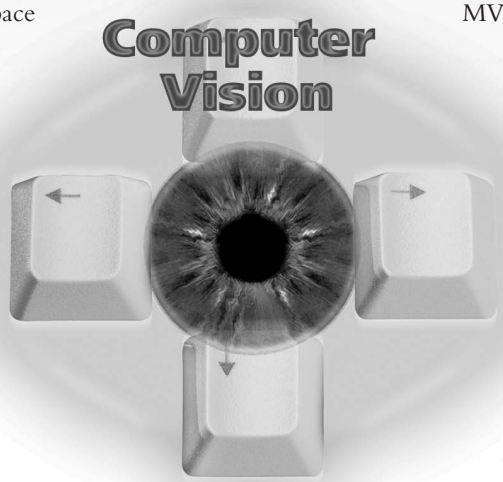
The Machine Vision Toolbox

A MATLAB Toolbox for Vision and Vision-Based Control

The Machine Vision Toolbox (MVT) described in this article provides many functions that are useful in machine vision and vision-based control. It is a somewhat eclectic collection reflecting the author's interest in areas of photometry, photogrammetry, and colorimetry. It includes more than 60 functions, spanning operations such as image file reading and writing, acquisition, display, filtering, blob, point and line feature extraction, mathematical morphology, homographies, visual Jacobians, camera calibration, and color space conversion. MVT, combined with MATLAB and a modern workstation computer, is a useful and convenient environment for investigation of machine vision algorithms. For modest image sizes, the processing rate can be sufficiently real time to allow for closed-loop control. Focus-of-attention methods such as dynamic windowing (not provided) can be used to increase the processing rate. With input from a firewire or Web camera (support provided) and output to a robot (not provided), it would be possible to implement a visual servo system entirely in MATLAB.



IRIS: ©1998 CORBIS CORPORATION, COMPUTER KEYS © DIGITAL VISION

An image is usually treated as a rectangular array of scalar values representing intensity or perhaps range. The matrix is the natural datatype for MATLAB and thus makes the manipulation of images easily expressible in terms of arithmetic statements in MATLAB language. Many image operations such as thresholding, filtering, and statistics can be achieved with existing MATLAB functions. MVT extends this core functionality with M-files that implement functions and classes and MEX files for some compute-intensive operations. It is possible to use MEX files to interface with image acquisition hardware ranging from simple framegrabbers to robots. Examples for firewire cameras under Linux are provided.

The routines are written in a straightforward manner, which allows for easy understanding. MATLAB vectorization has been used as much as possible to improve efficiency; however, some algorithms are not amenable to vectorization. If you have the MATLAB compiler available, then this can be used to compile bottleneck functions. Some particularly compute-intensive functions are provided as MEX files and may need to be compiled for the particular platform. MVT considers images generally as arrays of double precision numbers. This is extravagant on storage, though this is much less significant today than it was in the past.

MVT is not a clone of Mathwork's Image Processing Toolbox (IPT), although there are many functions in common. MVT predates IPT by many years, is open source, and contains many functions that are useful for image feature extraction and control. It was developed under Unix and Linux systems, and some functions rely on tools and utilities that exist only in those environments.

This article has a tutorial approach, illustrating the use of a subset of toolbox functions for some typical problems. First, file input and output, image acquisition, and display are described. A range of linear and nonlinear image operations, convolutions, and window operations follows, then a description of the extraction of blob, interest point, and line features. Next, a virtual camera is introduced and the effect of camera motion on image plane points is observed, bringing together many concepts, along with tools from the companion Robotics Toolbox [1] to simulate a complete image-based visual-servo system. Finally, conclusions and future work are discussed, describing how to obtain MVT.

BY PETER I. CORKE

1070-9932/05/$20.00©2005 IEEE

Input and Output

MATLAB now supports reading from a large number of file formats using `imread()`, but this was not always the case. MVT has functions for reading and writing files in PGM, PPM, and Inria file formats.

The function

```
>> lena = loadpgm ('lena');
>> idisp (lena)
```

will produce the display shown in Figure 1. Clicking on a pixel displays its coordinate and value in the top right corner; the image can be zoomed (and unzoomed), and the line function displays the intensity profile along a line between two clicked points. For multidimensional images, the pixel's vector value is displayed. A region-of-interest (ROI) can be simply extracted using the `iroi()` function, which can be interactive and return the region bounds as well as the region.

The functions `loadpgm()` and `loadppm()` will pop up a file selection graphical user interface (GUI) if no file is specified and support a local extension for timestamps embedded as header comments. If given a wildcard file definition, the functions will load a sequence of files:

```
>> seq = loadpgm('seq*.pgm');
>> size(seq)
ans =
      512   512 100,
```

in which case, the image is saved as `uint8` class to economize on storage. MPEG and AVI format files are most easily dealt with by saving individual frames as separate files in PPM or JPEG format and loading them individually. Tools such as `ffmpeg` [2] and `convert` [3] can easily perform this task. A group of functions allows reading uncompressed color image sequences in `yuv4mpeg` format.

Most Unix and Linux systems come with a wealth of quality image processing tools such as ImageMagick [3] and PBMplus [4]. We can exploit these using the `pnmfilt()` function, which effectively pipes the image through the external program. For example, we can rotate an image using an existing tool simply by using

```
>> rlena = pnmfilt(lena, 'pnmrotate 30');
>> image (rlena);
```

with the result shown in Figure 2. This function also allows us to access the huge range of file conversion filters that are a standard part of the PBMplus package. One of my favorite image viewing programs is `xv` [5], and we can display MATLAB images using

```
>> xv(lena),
```

which will create a subprocess and display the image in a new window.

Images can be read from a Web camera using

```
>> im = webcam('http://webcam7.cat.csiro.au');
```

or a firewire source (under Linux) using

```
>> fw = firewire(0, 'mono', 15);
  % open
  >> im = firewire(fw); % get next frame,
```

where the first call opens the firewire camera specified by its port in monochrome mode at 15 frames/s, and the handle is used in subsequent calls.

Image Operations

Given that an image is a matrix, we can perform very standard MATLAB matrix operations and display the results as an image. Examples include simple arithmetic, gamma correction

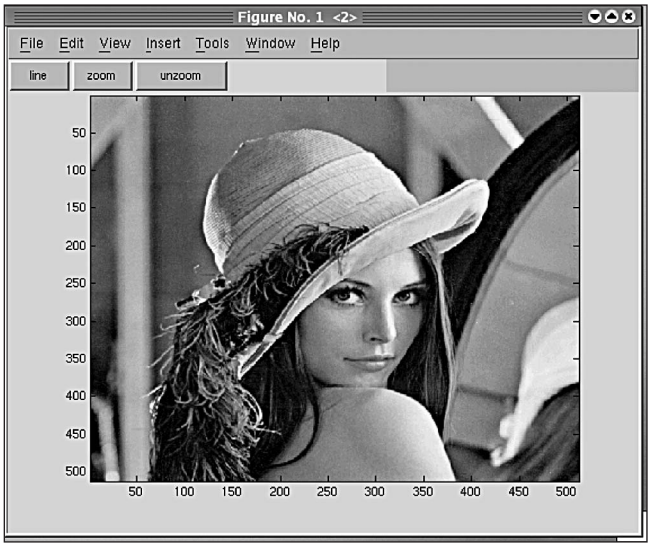


Figure 1. *Toolbox interactive display tool* `idisp()`.

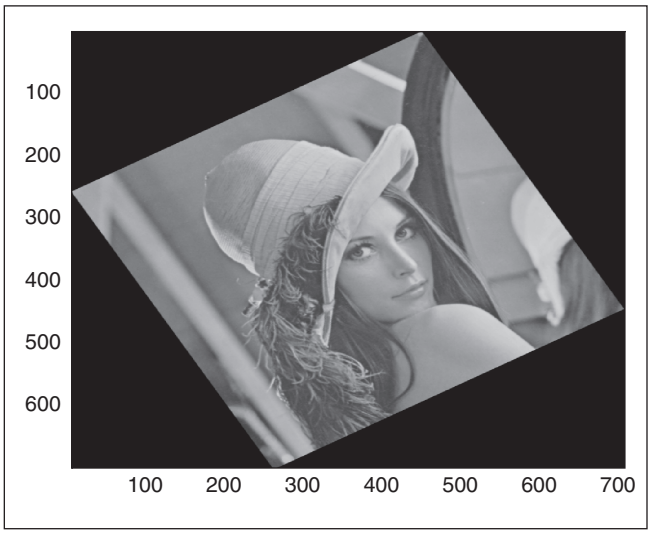


Figure 2. *Lena rotated by external PBMplus tool.*

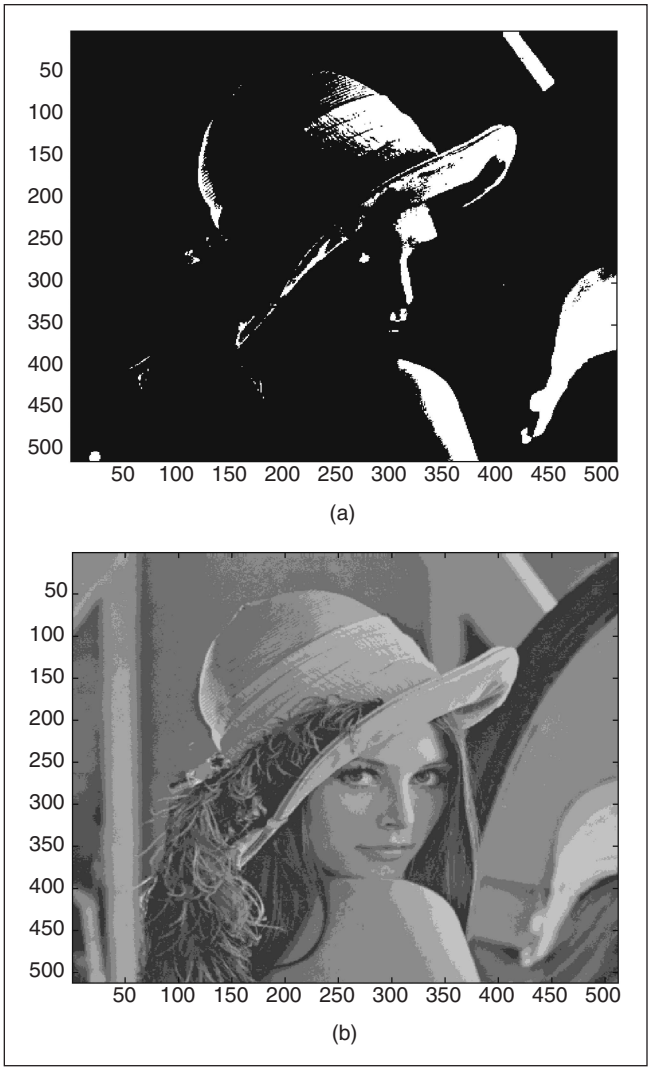


Figure 3. *Some simple nonlinear image operations: (a) thresholding and (b) solarization.*

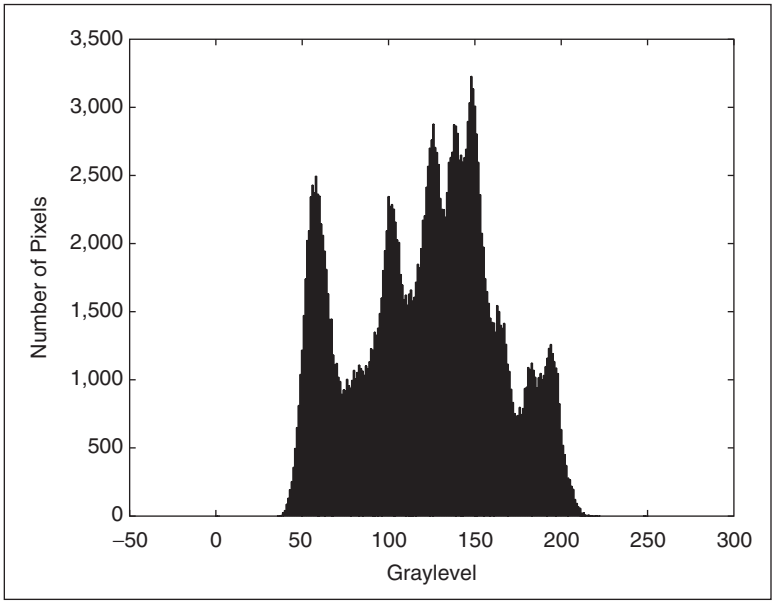


Figure 4. *A histogram of the Lena image.*

`igamma()`, greylevel normalization `inorm()`, and histogram normalization `inormhist()`. Nonlinear operations such as thresholding or solarization can be applied

```
>> idisp( lena > 180 )
>> idisp( bitand(lena, hex2dec('e0')),
```

and the results are shown in Figure 3.

The function `ihist()` provides a fast means of computing the histogram of a scalar-valued image

```
>> ihist( lena )
```

(see Figure 4) and can optionally return the histogram and bin center values. The histogram can be used to select optimal threshold values.

Window Operations

Some of the most powerful and commonly used image processing operations are convolutions, and these are provided in MATLAB by the `conv2()` function. Convolution kernels can be produced using toolbox functions such as Gaussian `kgauss()`, derivative of Gaussian `kdgauss()`, Laplacian `klaplace()`, Laplacian of Gaussian `klog()`, and difference of Gaussian `kdog()`. The functions

```
>> idisp(conv2(lena, kdgauss(1)));
>> idisp(conv2(lena, kdgauss(4)));
>> idisp(zcross(conv2(lena, kdgauss(4))));
>> idisp(isobel(lena));
```

are shown in Figure 5. The function `isobel()` computes the Sobel image operator, and `zcross()` finds zero crossings, i.e., pixels with neighbors of a different sign.

Frequently, it is useful to reduce the resolution of an image by smoothing and subsampling. The pyramid function

```
>> p = ipyramid(lena);
>> p'
ans =
 [512x512 double]
 [256x256 double]
      .
      .
 [  4x4 double]
 [  2x2 double]
 [     20.8959]
>> idisp(p {5})
```

returns a MATLAB cell array containing images that are at successively lower resolutions. The fifth element of the pyramid is shown in Figure 6. Note that the last element is the 1×1 resolution version.

A generalized window operation is provided by `iwindow()`. For every pixel in the input

image, it takes all neighbors for which the corresponding element in the structuring element are nonzero. These are packed into a vector (in raster order from top left) and passed to the specified MATLAB function. The return value becomes the corresponding output pixel value. To compute the mean of an image over an annular window at each point:

```
>> % inner/outer radius = 5/10
>> se = kcircle([5 10]);
>> out = iwindow(lena, se, 'mean');
```

is used.

The structuring element is a matrix with odd dimensions that is centered on the output pixel. Square neighborhoods can be specified conveniently using `ones(N,N)`, circular neighborhoods using `kcircle(R)`, and annular neighborhoods using `kcircle(R1, R2)`. Boundary handling can be controlled by an optional fourth argument.

Similar functions to compute statistics within the selected pixels are `ivar()`, which can compute variance, kurtosis, and skew, and `irank()`, which can return the *j*th ranked pixel (on intensity).

```
>> out_var = ivar (lena, ones(3,3), 'var');
>> out = irank(lena, ones(5,5), 1);
```

The first command computes the variance over a 3×3 window at every pixel (which is a reasonable edge operator); the last argument can also be `kurt` or `skew` for higher-order statistics such as kurtosis and skew. The second command returns the maximum (rank 1) pixel value over a 5×5 window at every pixel.

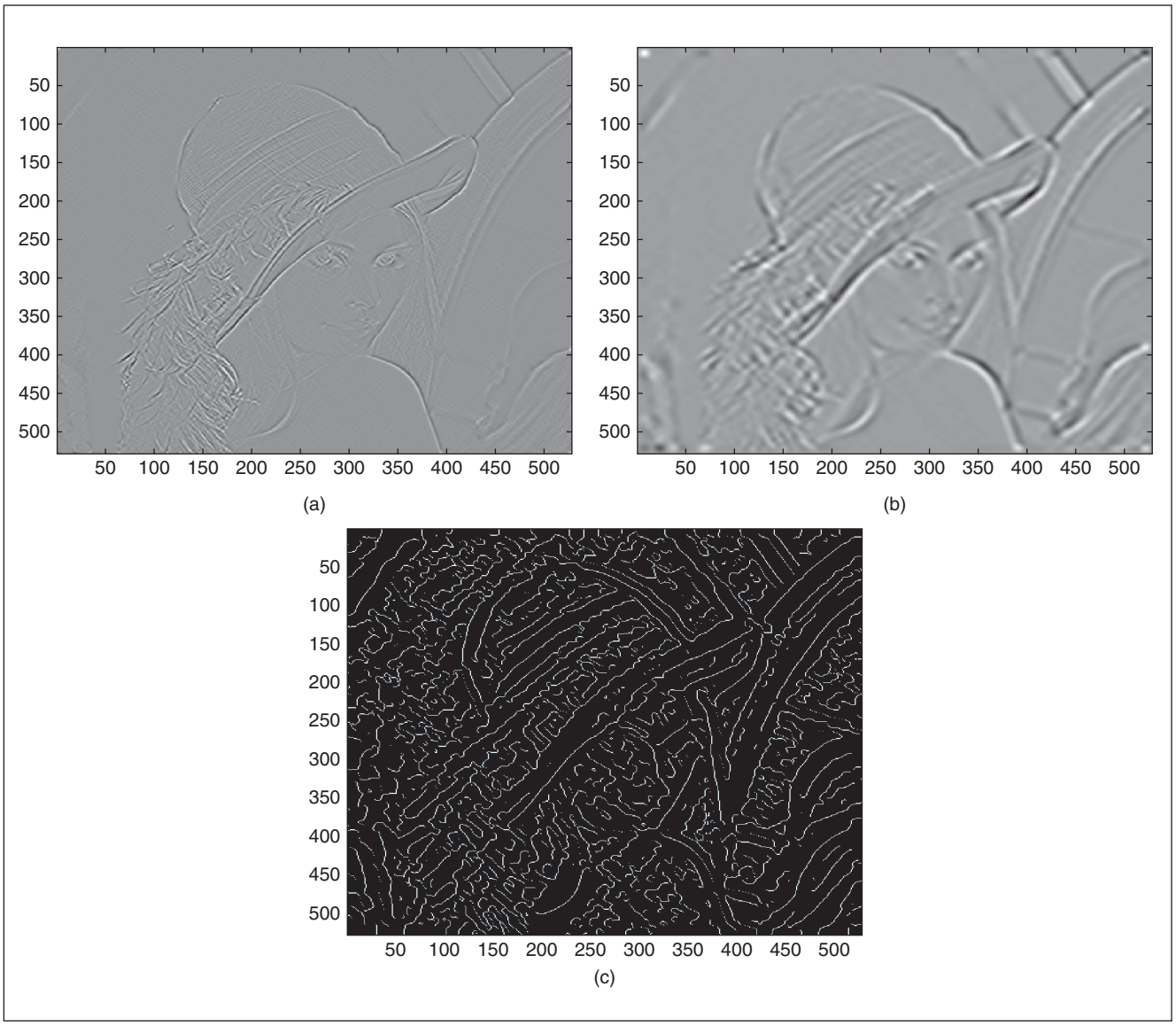


Figure 5. *Edge operators: (a) a derivative of Gaussian with $\sigma = 1$; (b) a derivative of Gaussian with $\sigma = 4$; and (c) zero crossings of derivative of Gaussian with $\sigma = 4$.*

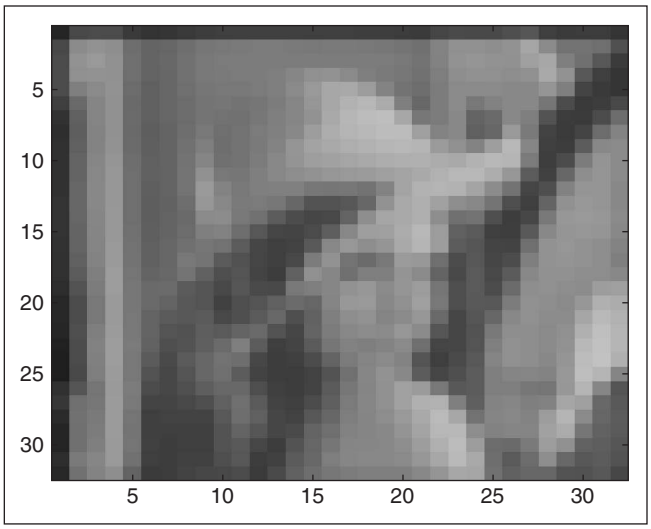


Figure 6. *Reduced resolution image at pyramid level 5.*

Image Feature Extraction

For vision-based control, it is critical to extract features from the image that are used as input to the visual control law. The features used are typically points, which can be the centroids of regions, interest operators, or lines.

Blob Features

Segmentation is the process of dividing an image into meaningful segments, generally homogeneous with respect to some characteristic. In a simple or contrived scene, the segments may correspond directly to objects in the scene, but for a complex scene, this will not necessarily be the case. The simplest classification is into two sets, leading to binary segmentation. Commonly, this is achieved by applying a threshold test to the pixel values. Consider the image in Figure 7, which shows a scene containing four targets. We can segment these targets using

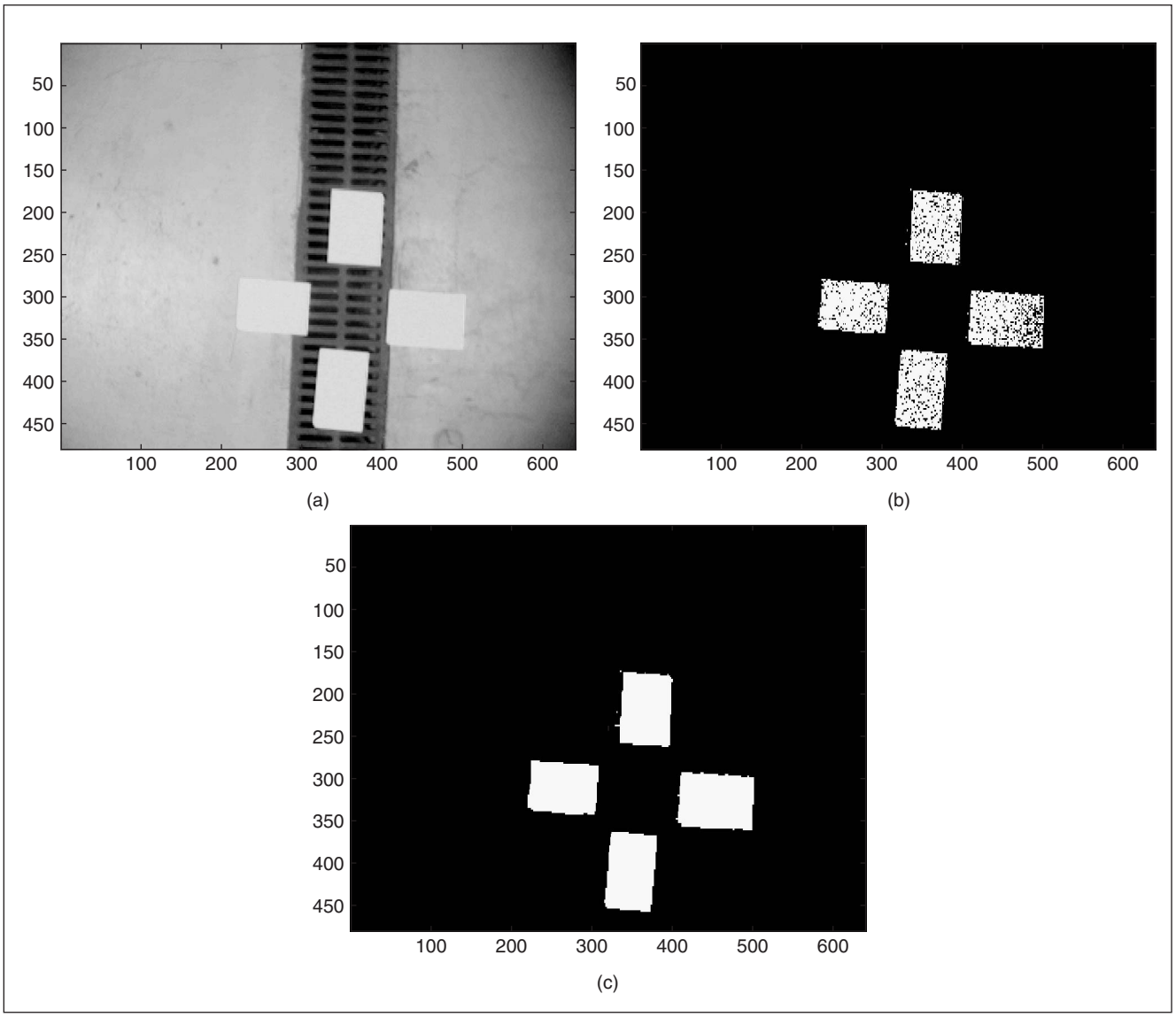


Figure 7. *Color image segmentation: (a) original color scene (the four boxes, i.e., targets, would appear in color); (b) initial segmentation based on bivariant histogram; and (c) final segmentation after morphological filtering.*

```
>> out = colorseg(target, lut);
>> idisp(out),
```

where `lut` is the result of a training process carried out using the interactive function `trainseg()`. Segmentation is based on red-green chromaticity coordinates. Clearly, the resulting segmentation leaves something to be desired, as noise has caused various target pixels to be misclassified as background.

We can apply mathematical morphology techniques to the binary image in order to fill these holes. Mathematical morphology is a set of techniques for filtering objects in an image based on their size and shape. The operation performed on the pixels selected is either the minimum, maximum, or difference (maximum-minimum) as specified by the third argument, which is either `min`, `max`, or `diff`, respectively. Boundary handling flags, as mentioned previously, can also be applied.

In this case, we wish to take the maximum value in a 5×5 region around each pixel.

```
>> clean = imorph(out, ones(5,5), 'max');
>> idisp(clean)
```

The function `iclose()` performs N maximum operations followed by N minimum operations and is used for filling interior holes while leaving the outer dimensions the same.

Now that we have a clean binary image, we can label the various connected components.

```
>> labels = ilabel(clean);
>> idisp(labels)
```

Figure 8 shows the results. Each region has been assigned a unique integer label, which is shown as a unique gray level. More often, we want to label the image, compute blob features, and possibly apply some filtering or selection criteria.

```
>> F = iblobs(clean, 'area', [100 10000]);
>> F
F =
1x4 struct array with fields:
    area xc yc a b theta m00 m01 m10
    m02 m20 m11 minx maxx miny maxy touch
```

In this instance, `F` is a four-element structure array, one per target object. The filter applied is for area A such that $100 < A < 10,000$. Each element contains the centroid, moments, bounding box, and edge touch flag. The centroids

```
>> [ [F.xc]' [F.yc]' ]
ans =
  366.8313 218.5545
  264.8579 310.9883
  453.5889 326.7215
  348.2511 409.1840
>> hold on
>> for i=1:length(F)
>> plot(F(i).xc, F(i).yc, 'g+')
>> end
```

are shown by the overlaid crosses in Figure 8.

Point Features

The Harris interest point detector [6] is a classic technique to robustly detect in a scene points that have strong orthogonal image gradients. Such points are more likely to be tracked across an image sequence.

To compute and overlay the detected corners on a street scene,

```
>> C = iharris(building);
>> idisp(building);
>> markfeatures(C, 50, 'wo');
```

will display the 50 strongest corners as white circles (see Figure 9). The function `markfeatures()` with output arguments will be assigned the pixel coordinates of the corner

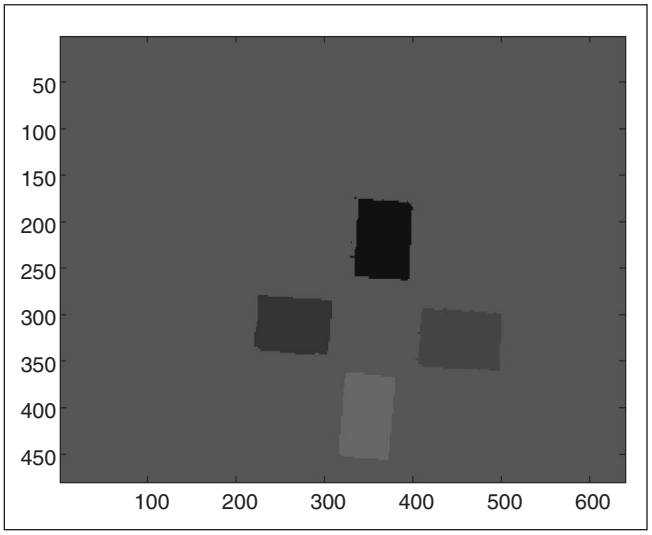


Figure 8. *Labeled image with centroids marked.*

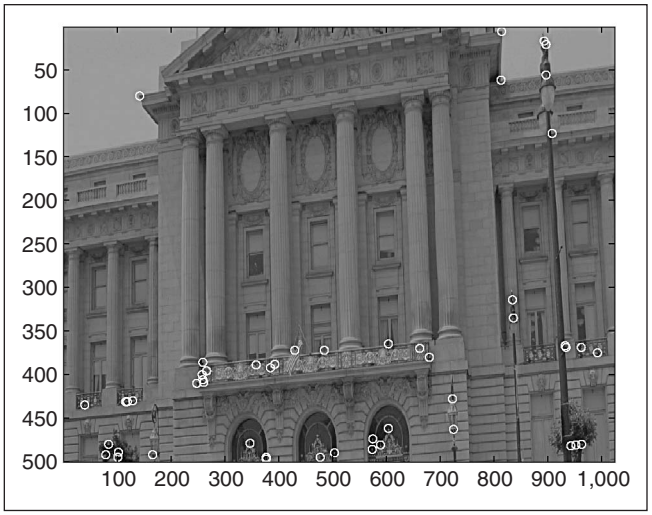


Figure 9. *Harris corner features.*

MVT, combined with MATLAB and a modern workstation computer, is a useful and convenient environment for investigation of machine vision algorithms.

feature. The `iharris()` function accepts a parameter structure that allows control over the derivative kernel used, minimum corner strength, and distribution of corners across the image, which, as can be seen in Figure 9, tend to cluster unevenly in the image.

Line Features

The Hough transform is a classical technique to identify dominant lines in a scene. All lines are represented in a d-θ parameterization

$$d = y\cos(\theta) + x\sin(\theta), \tag{1}$$

where θ is the angle the line makes to the horizontal axis and d is the perpendicular distance between (0,0) and the line. We quantize the d-θ space, and for every detected edge pixel, we vote for all possible lines that pass through that point. The peaks in the voting accumulator correspond to dominant lines.

To compute the edges, see Figure 10 and the Hough accumulator

```
>> building = imono( imread('building.jpg') );
>> edges = isobel(building);
>> H = ihough(edges).
```

The accumulator array can be displayed

```
>> houghshow(H);
```

as an image that shows bright spots that correspond to dominant edges. To find the peaks and the corresponding lines in the original image, use

```
>> p = houghpeaks(H, 10);
>> idisp(building)
>> houghoverlay(p);
```

where `p` is a matrix with one row per dominant line and the columns are d and θ, respectively. Results for the building scene are shown in Figure 10.

Virtual Cameras

MVT implements a virtual camera, for example

```
>> cp = pulnix
>> C = camcalp(cp)
>> cam = gcamera('my camera', C, [0 512 0 512]),
```

where `cp` is a set of parameters for a particular camera and lens [7], which includes pixel pitch, focal length, and principal point. The function `camcalp()` converts the parameter vector into a camera projection matrix and can take an extra argument to specify the pose of the camera; by default it is assumed to be at the origin with its optical axis pointing in the z-direction. The routine `gcamera()` creates a window that displays the image plane of the virtual camera and has parameters that define the projection matrix and the coordinate system of the image plane. Multiple cameras can be instantiated to simulate a stereo or trinocular camera system, for instance. The routine `gcamera()` is a wrapper around the non-GUI function `camera()`.

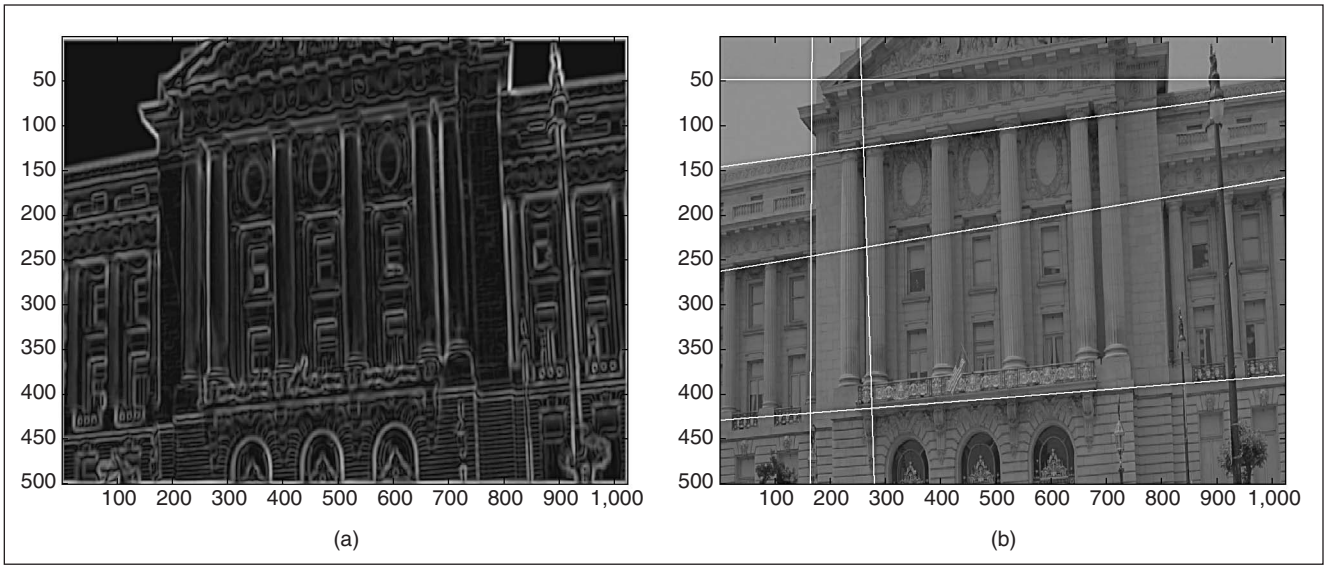


Figure 10. *Hough transform of a building: (a) building scene edge image and (b) building scene with overlaid dominant lines.*

Consider a target defined in the XY plane as

```
>> targ = [-0.5 -0.5 0; -0.5 0.5 0; ... 0.5
   0.5 0; 0.5 -0.5 0];
>> uv = gcamera(cam, targ, transl(0,0,5) )
   uv =
     150.2550      21.7263
     150.2550     398.2737
     397.7450     398.2737
     397.7450      21.7263,
```

then we can view it with the target translated by 5 m in the z-direction (see Figure 11). Note that in the example given, we can see that the pixels are not square, and the image looks compressed in the horizontal direction. If we rotate the target around its y-axis, we observe significant foreshortening, as shown in Figure 12. The functions `roty` and `transl` are from the companion Robotics Toolbox [1].

```
>> uv = gcamera(cam, targ, ...
    transl(0,0,5)*roty(1) )
uv =
   243.1686   123.1802
   243.1686   296.8198
   310.5066   312.8008
   310.5066   107.1992.
```

In the example given, we have computed the projection matrix from a set of camera parameters, but in practice, we often do not know these parameters; they need to be determined experimentally, a procedure known as camera calibration. MVT provides several functions to estimate this matrix from world and image-plane coordinates: the least-squares method of Sutherland [8] `camcald()`, which requires a three-dimensional (3-D) target, and the method of Tsai [9] `camcalt()`, which requires a planar target and lens distortion. Other toolboxes available on the Web provide a more comprehensive treatment of camera calibration [10].

MVT has functions for reading and writing files in PGM, PPM, and Inria file formats.

Image Motion

Let's examine the motion of points on the image plane as we move the camera slightly.

```
>> uv1 = gcamera(cam, targ, transl(0,0,5) )
>> uv2 = gcamera(cam, targ, transl(0,0,5), ...
            transl(0.1, 0.2, 0)*roty(0.2) )
```

Since our target is planar, we can compute a matrix, `H`, which is an homography, i.e., a homogeneous transformation that maps `uv1` to `uv2`.

```
>> H = homography(uv1, uv2)
H =
   1.2030   0.0000 -180.5223
   0.0761   1.1262  -68.9098
   0.0004 -0.0000    1.0000
>> uv2 - homtrans(H,uv1)
ans =
```

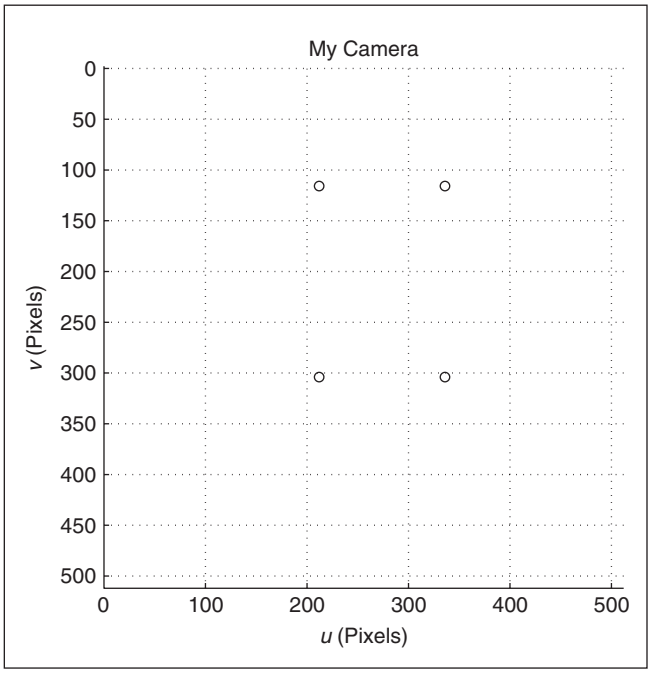


Figure 11. *Virtual camera view with the target normal to the optical axis.*

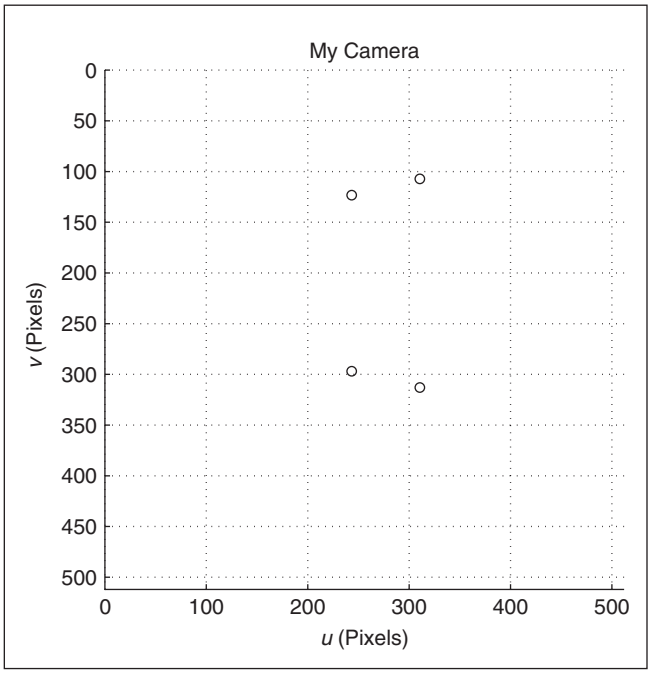


Figure 12. *Virtual camera view, with the target rotated about the y-axis.*

```
1.0e-08 *
 0.0485   -0.0209
 0.0164   -0.2033
-0.0448   -0.2175
 0.0045   -0.0448
```

The relationship between small camera motion and image plane feature motion $(\dot{u}, \dot{v})$ is given by the image Jacobian matrix

$$\begin{bmatrix} \dot{u} \\ \dot{v} \end{bmatrix} = \mathbf{J} \begin{bmatrix} T_x \\ T_y \\ T_z \\ \omega_x \\ \omega_y \\ \omega_z \end{bmatrix}, \tag{2}$$

where

$$\mathbf{J} = \begin{bmatrix} \frac{\lambda}{z} & 0 & \frac{-u}{z} & \frac{-uv}{\lambda} & \frac{\lambda^2+u^2}{\lambda} & -v \\ 0 & \frac{\lambda}{z} & \frac{-v}{z} & \frac{-\lambda^2-v^2}{\lambda} & \frac{uv}{\lambda} & u \end{bmatrix}. \tag{3}$$

The function `visjac_p()` computes a 2×6 visual motion Jacobian that maps relative camera motion $[T_x, T_y, T_z, \Omega_x, \Omega_y, \Omega_z]$ to image plane velocity for point features. Jacobians for line and circle features are also provided. For three or more image points, the Jacobians can be stacked and, if full rank, we can solve for the camera velocity needed to achieve arbitrary image point velocity—the essence of image-based visual servoing (IBVS) [11].

IBVS

In this section, we bring together a number of functions from MVT and from the companion Robotics Toolbox [1]. The Simulink model, `demo6`, shown in Figure 13, demonstrates IBVS [11]. This is quite a complex example that simulates not only the robot but also a camera and the IBVS algorithm. The camera is assumed to be mounted on the robot's end effector, and this coordinate is passed into the camera block so that the relative position of the target with respect to the camera can be computed. Arguments to the camera block include the 3-D coordinates of the target points. The output of the camera is the two-dimensional (2-D) image plane coordinates of the target points. The target points are used to compute an image Jacobian matrix, which is inverted and multiplied by the desired motion of the target points on the image plane. The desired image plane motion is simply the difference between the observed target points and the desired point positions. The result is a velocity screw that drives the robot to the desired pose with respect to a square target.

To run this demo, use

```
>> ibvsdemo.
```

When the simulation starts a new window, the camera view pops up. We see that initially, the square target is off to one side and somewhat oblique. The image plane errors are mapped by an image Jacobian into desired Cartesian rates, and these are further mapped by a manipulator Jacobian into joint rates, which are applied to the robot, which is again modeled as a rate control device. A Puma 560 robot is used in this simulation.

This closed-loop system is performing a Cartesian positioning task with information from a camera rather than with encoders and a strong geometric and robot kinematic

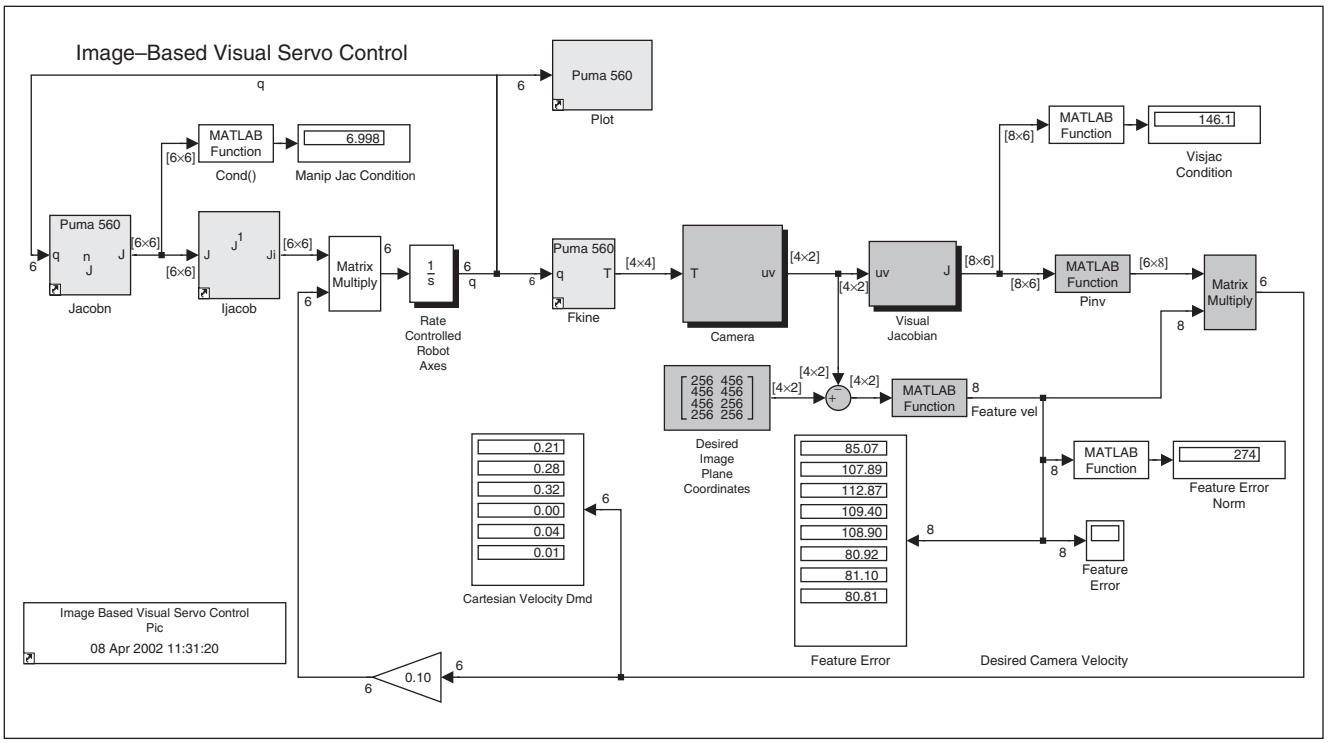


Figure 13. *Robotics Toolbox example* `demo6`*, image-based visual servoing.*

model (the Jacobian is a weak kinematic model). Image-based visual servoing schemes have been found to be extremely robust with respect to errors in the camera model and manipulator Jacobian, and the Simulink environment is an ideal one in which to explore this. While not demonstrated by the author, others have used MEX files to interface a MATLAB/Simulink controller to a real physical robot. Combined with the Firewire camera interface, `firewire()`, a complete visual servo system could be implemented in MATLAB, albeit with a sample rate slower than could be achieved if the code were written in C.

Miscellaneous

MVT contains a number of functions that are difficult to otherwise categorize and that reflect the author's personal interest in areas such as photometry, photogrammetry, and colorimetry. This includes functions for the spectral characteristics of blackbody radiators, the photopic response of the human eye, and conversions between color spaces such as RGB, HSI, and CIE XYZ.

The `testpattern()` function can generate images with a variety of patterns including lines, grids of dots or squares, intensity ramps, and intensity sinusoids.

Conclusions

This article has described an open-source toolbox for machine vision that can be used for research into machine vision but is also versatile enough to be usable for real-time work and even control. However, its use for control is very system specific and requires a high level of user expertise and experience, particularly when interfacing to a robot.

MVT is a personal collection of functions that has grown over more than a decade. Naturally, for something that has grown organically, there are things that could have been done better, but often these design decisions were due to the functionality of MATLAB at that time. Long ago, numeric classes and objects did not exist. The biggest problems, in the author's opinion, are inconsistent naming patterns for functions, patchy support for images represented in the compact `uint8` class, and the treatment of images' pixel values in the range 0 to N (typically $N = 256$) rather than normalized as 0 to 1. Multidimensional images could represent an image sequence or an image with vector-valued pixels. MVT has no mechanism to distinguish these cases or to handle a sequence of vector-valued images.

As with all open-source software, if it helps you, then use it. If you need to change it, then do so, and maybe send the changes back to the author. If it doesn't meet your needs, then move on.

Obtaining MVT

MVT, and the Robotics Toolbox, can be obtained from http://www.cat.csiro.au/ict/staff/pic. Most functions are implemented as M-files, and many of these should operate with the open-source package Octave [12] when it is set to operate in MATLAB compatibility mode. Handle-graphics functions will not work with Octave. More compute-intensive functions are coded in C as MEX files and must be compiled before they can be used—these have been tested under many variants of Linux and Mac OS X, but not Windows.

Acknowledgments

Some MEX files are based on code that was part of the package VISTA (copyrighted 1993 and 1994 by the University of British Columbia, Canada). Homography routines are based on code by Nuno Alexandre Cid Martins, Universidade de Coimbra, Portugal.

Keywords

Machine vision, MATLAB, visual servo, image filtering, image segmentation, image feature extraction.

References

[1] P. Corke, "A robotics toolbox for matlab," *IEEE Robot. Automat. Mag.*, vol. 3, pp. 24–32, Sept. 1996.

[2] "ffmpeg multimedia system" [Online]. Available: http://ffmpeg.sourceforge.net/index.php

[3] "Imagemagick," ImageMagick Studio LLC [Online]. Available: http://www.imagemagick.org/

[4] J. Poskanzer, "Pbmplus" [Online]. Available: http://www.acme.com/software/pbmplus/

[5] J. Bradley, "Xv interactive image manipulation program" [Online]. Available: http://www.trilon.com/xv/

[6] C.G. Harris and M.J. Stephens, "A combined corner and edge detector," in *Proc. 4th Alvey Vision Conf.*, Manchester, England, 1988, pp. 147–151.

[7] P.I. Corke, *Visual Control of Robots: High-Performance Visual Servoing*. New York: Wiley, 1996.

[8] I.E. Sutherland, "Three-dimensional data input by tablet," *Proc. IEEE*, vol. 62, pp. 453–461, Apr. 1974.

[9] R. Tsai, "An efficient and accurate camera calibration technique for 3D machine vision," in *Proc. IEEE Conf. Computer Vision Pattern Recognition*, 1986, pp. 364–374.

[10] J.-Y. Bouguet, "Camera calibration toolbox for MATLAB" [Online]. Available: http://www.vision.caltech.edu/bouguetj/calib_doc

[11] S. Hutchinson, G. Hager, and P. Corke, "A tutorial on visual servo control," *IEEE Trans. Robot. Automat.*, vol. 12, pp. 651–670, Oct. 1996.

[12] J.W. Eaton, "Octave language for numerical computations" [Online]. Available: http://www.octave.org/

Peter I. Corke is the research director of the Autonomous Systems Laboratory within the CSIRO Information and Communications Technology (ICT) Centre, in Brisbane, Australia. His research activities span machine vision, vision-based robot control, field robotics (with a focus on mining applications), and sensor networks. He holds B.Eng. and M.Eng.Sc. degrees, both in electrical engineering, and a Ph.D., all from the University of Melbourne, Australia. He is an adjunct professor at the Australian National University and the University of Queensland and is a member of the editorial board of the *International Journal of Robotics Research*.

Address for Correspondence: Peter Corke, CSIRO ICT Centre, P.O. Box 993, Kemore 4069 Brisbane, Australia. E-mail: peter.corke@csiro.au.